



IBM Research

# The Complexity of Andersen's Analysis in Practice

Manu Sridharan and Stephen J. Fink  
IBM T.J. Watson Research Center

SAS 2009

## Andersen's Analysis

- **Definition (almost)**  
*precise flow- and context-insensitive points-to analysis*
- Presented by Andersen in 1994
  - Similar predecessors, e.g., 0-CFA [Shivers88]
- Worst-case complexity ***nearly cubic*** (  $O(N^3 / \log N)$  )
- Early implementations didn't scale
  - Approximations developed [Steensgaard96, Das00]

## Scaling Andersen's Analysis

**Online Cycle Elimination**  
[FFSA98,HT01,HL07]

**Type Filters**  
[LH03]

**Preprocessing**  
[RC00,HL07]

**Shared Bitsets / BDDs**  
[HT01,BLQHU03,ZC04,WL04]

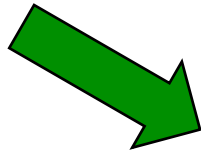
**Projection Merging**  
[SFA00]

**And More!**  
[next talk]

**Impressive Scalability:** 1M C LOC [HL07] or 500K  
Java bytecodes [WL04] in under 10 minutes

## Why Does Andersen's Scale?

Our work,  
for Java



### Possibility 1: Reduced Constant Factors

Nearly cubic behavior remains in practice

### Possibility 2: Real Programs Easier

Program structure enables *subcubic scaling* in practice

## Key Results

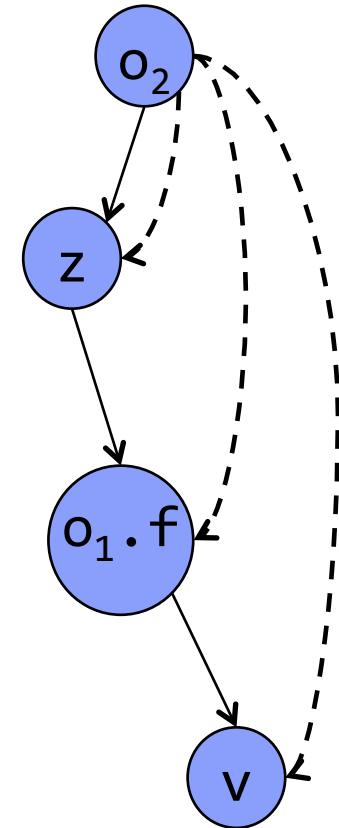
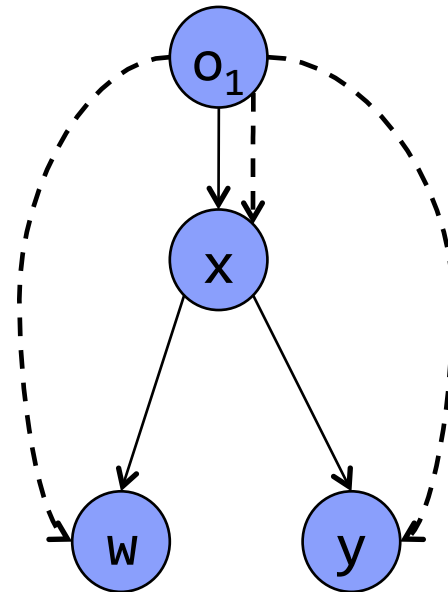
- Andersen's analysis is  **$O(N^2)$**  for ***k*-sparse programs**
- For Java, *k*-sparsity through **types + encapsulation**
  - Structure makes analysis easier than for C
- **Empirical validation**
  - Benchmarks from 176-2225K bytecodes
  - Showed *k*-sparsity and quadratic scaling

# Background: Andersen's as Dynamic Transitive Closure

```

1: x = new Obj();
2: z = new Obj();
3: w = x;
4: y = x;
5: y.f = z;
6: v = w.f;

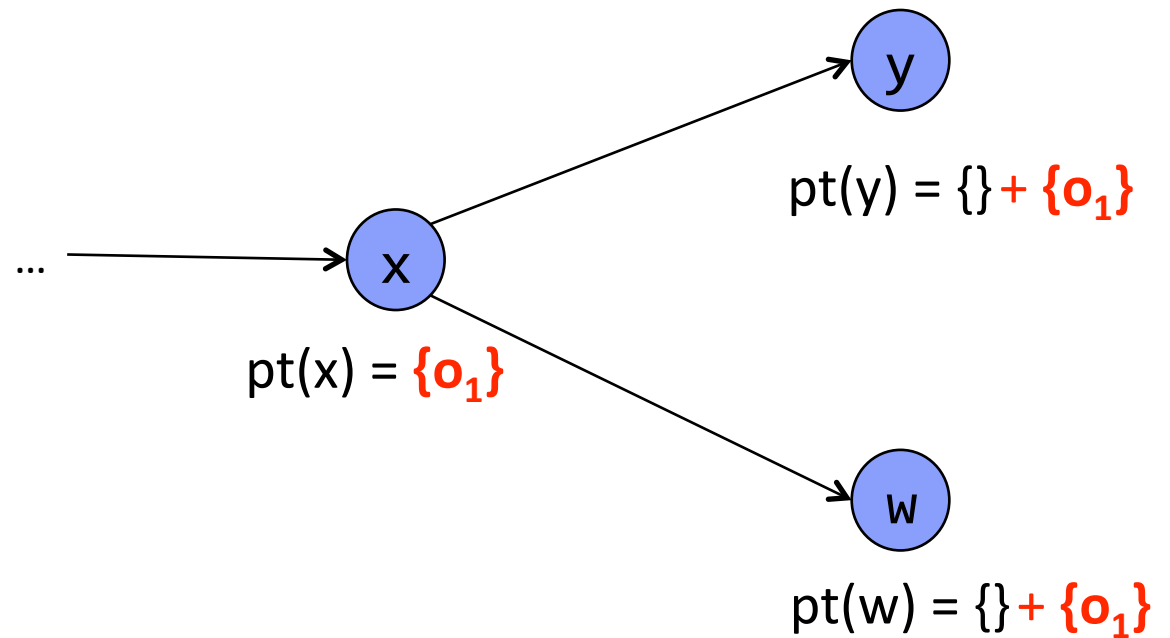
```



Complexity of chaotic  
worklist algorithm:  **$O(N^4)$**

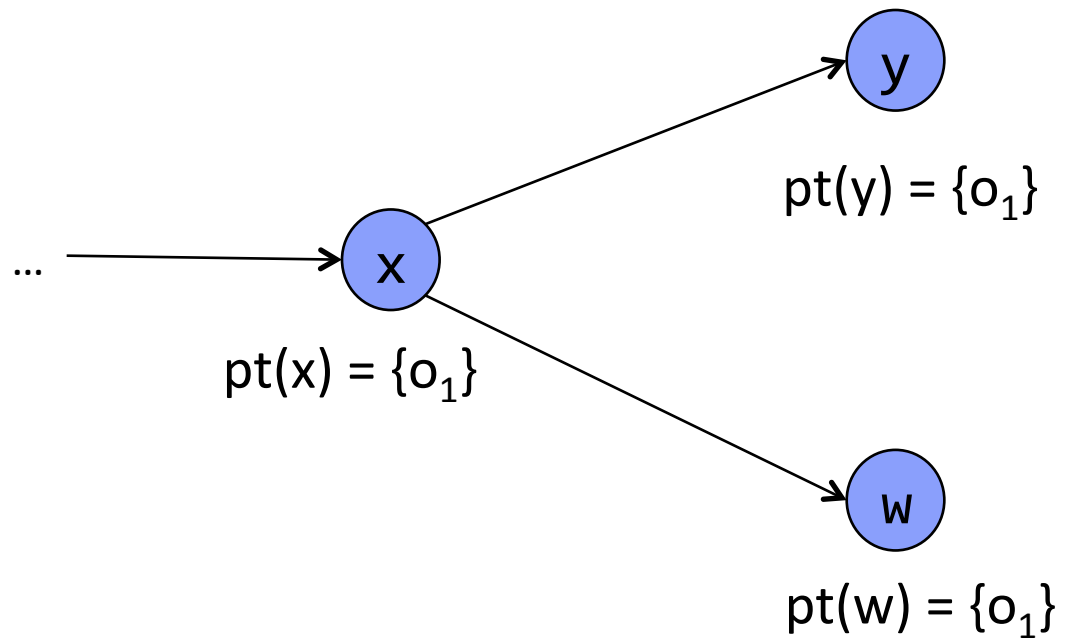
## Background: Difference Propagation

“Standard”  
Propagation



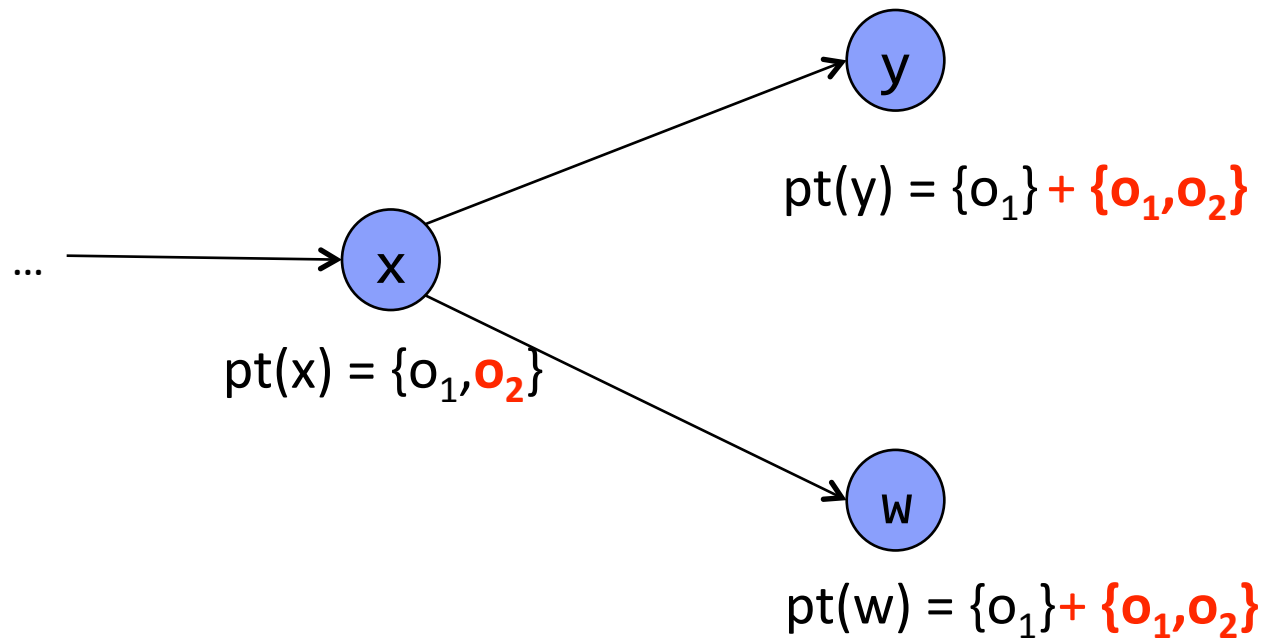
## Background: Difference Propagation

“Standard”  
Propagation



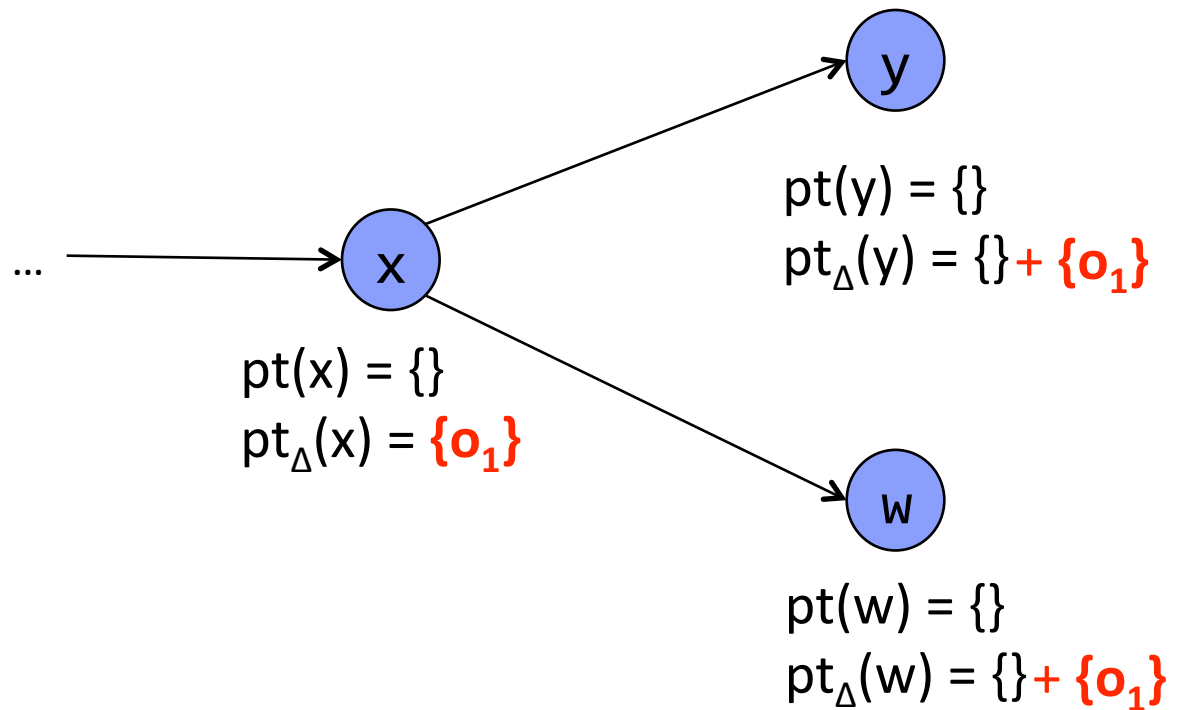
## Background: Difference Propagation

“Standard”  
Propagation



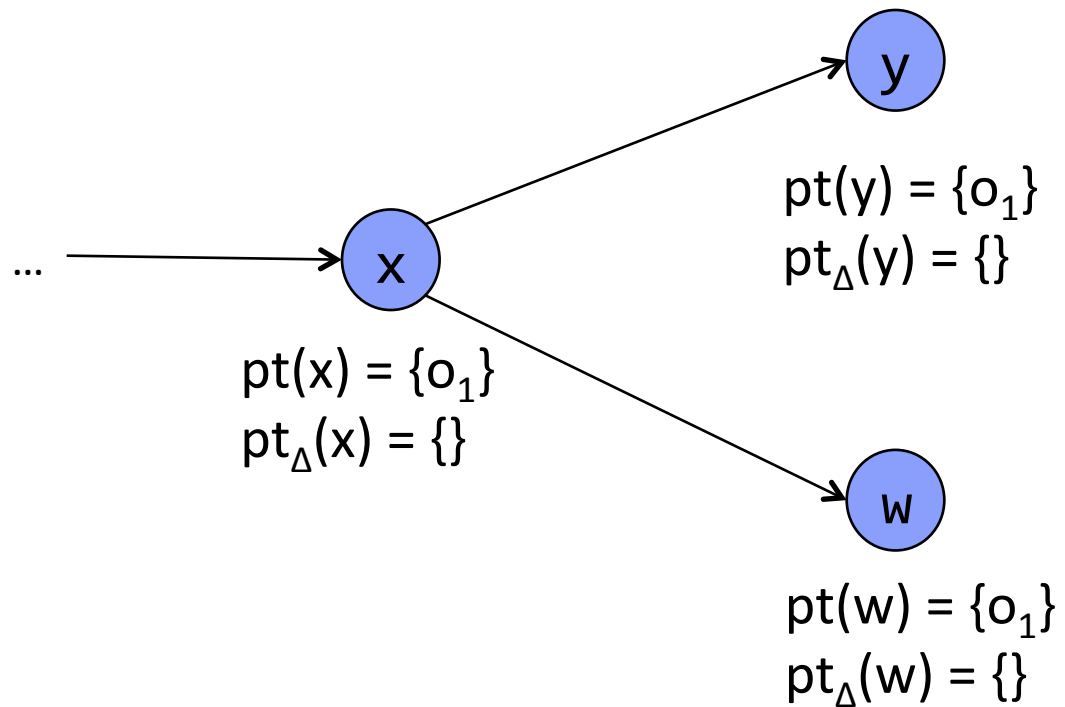
# Background: Difference Propagation

## Difference Propagation



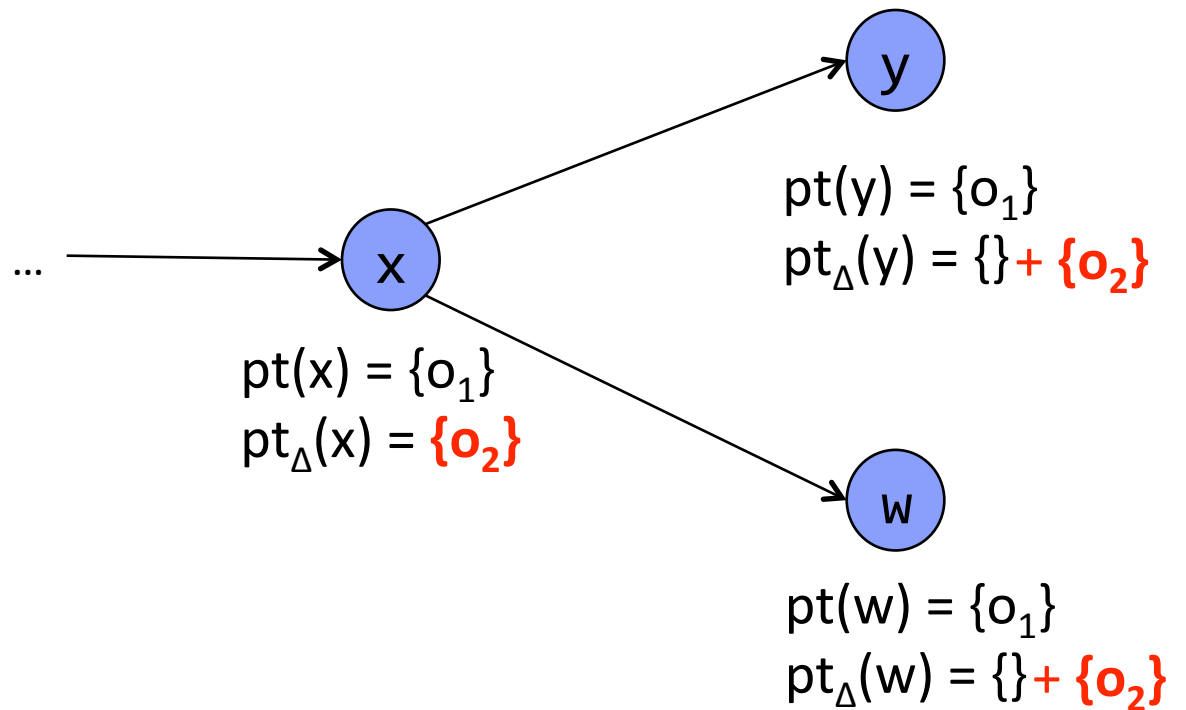
# Background: Difference Propagation

## Difference Propagation



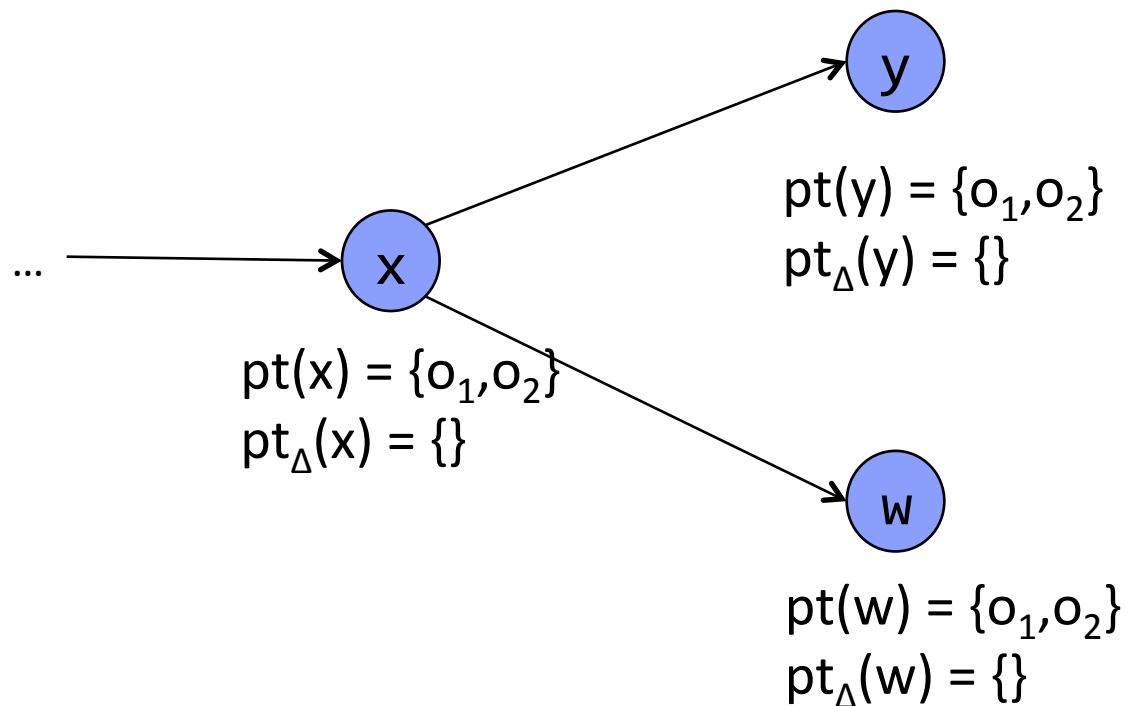
# Background: Difference Propagation

## Difference Propagation



## Background: Difference Propagation

### Difference Propagation

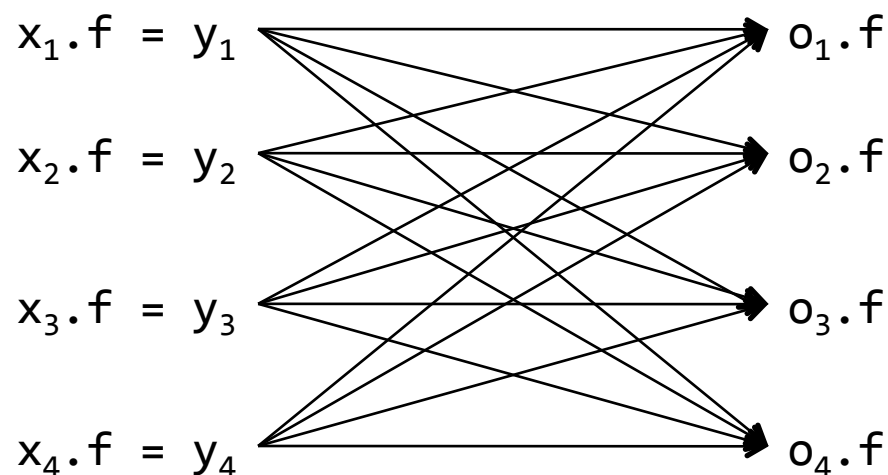


- Guarantee: loc propagated **at most once per edge**
- DTC + difference propagation complexity:  $\mathbf{O(N^3)}$  [Pearce05]

## *k*-sparse programs

- Def: num. of graph edges  $\leq k * N$  at termination,  $k$  constant
- Complexity for  $k$ -sparse programs:  **$O(N^2)$** 
  - Linear number of edges, linear work per edge (via diff. prop.)
  - Must also count edge adding work; see paper for details

### Non- $k$ -sparse graph



## Java and *k*-sparsity: *strong types*

```
class A { int f; }  
class B { int g; }
```

```
A a = new A();
```

```
a.g = 5; // compile error
```

### Key benefits: few fields per object, no aliased fields

- Limits number of object field nodes created
- Exploited in previous work [SGSB05,SB06]

Unlike C: no structure casts

## Java and $k$ -sparsity: *encapsulation*

```
class C {  
    // encapsulated  
    private int state;  
    int getState() { return this.state; }  
    void setState(int i) { this.state = i; }  
}
```

### Benefit: few accesses per field

- Limits number of closure edges
- Tradeoff: possibly worse precision (context insensitivity)

Unlike C: no \* operator

## Threats to $k$ -sparsity

- Dynamic dispatch
  - # of targets at call sites may increase with program size
  - Haven't observed in practice; on-the-fly call graph helps
- Arrays
  - $y = x[0]; \rightarrow y = x.arr;$
  - Same `arr` field for all array types (due to subtyping)
  - # of accesses of `arr` increases with program size (like `*` in C)
  - Observed some blowup in one benchmark

# Experiments

## Implementation

T. J. Watson Libraries  
for Analysis (WALA)

<http://wala.sf.net>

## Benchmarks

Dacapo 2006-10-MR2 + Apache Ant

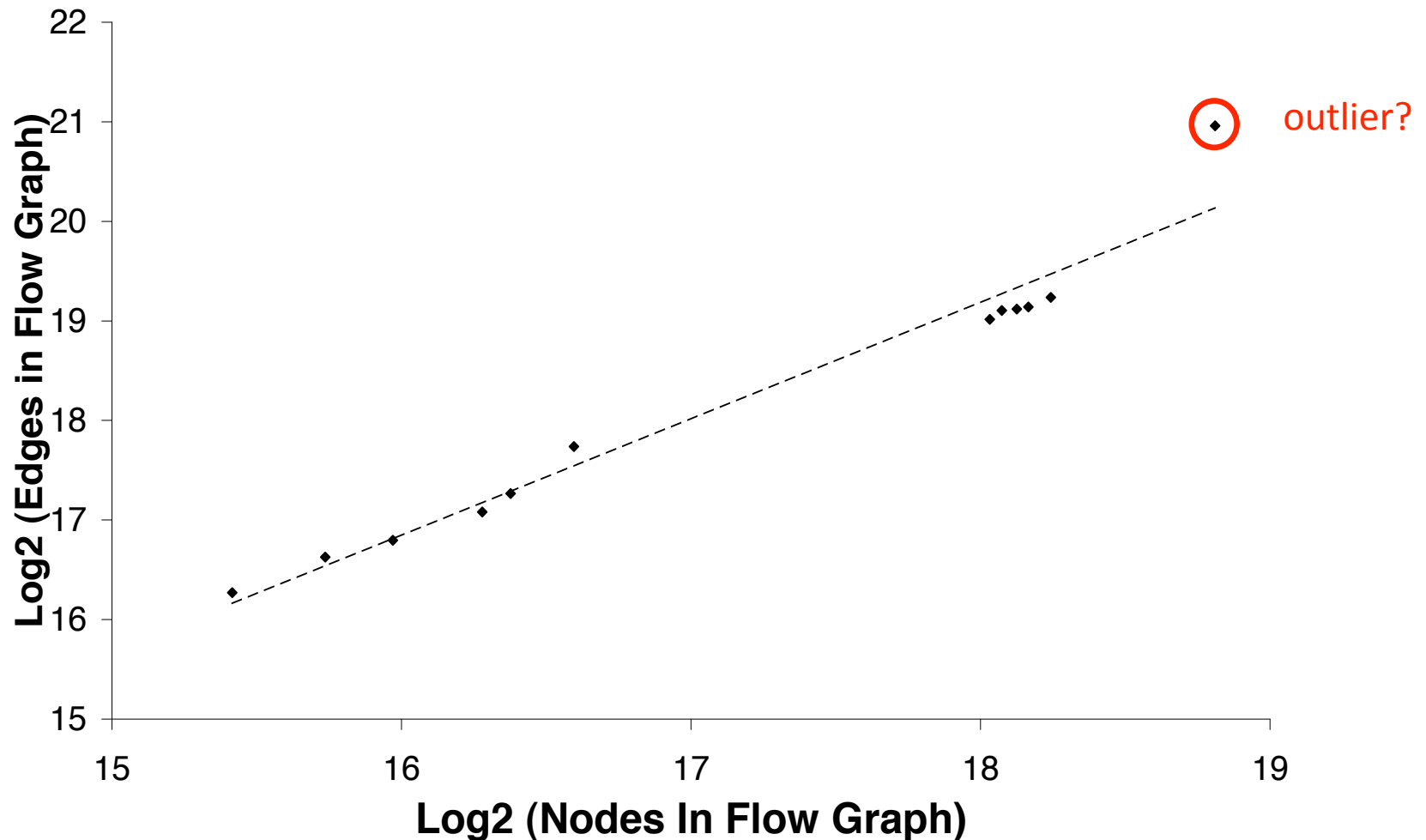
IBM Java 1.6.0 libraries

176-2225K bytecodes (largest published)

## Questions

1. Are programs  $k$ -sparse?
2. Is quadratic scaling observed?
3. How tight is quadratic bound?

## Are programs $k$ -sparse?



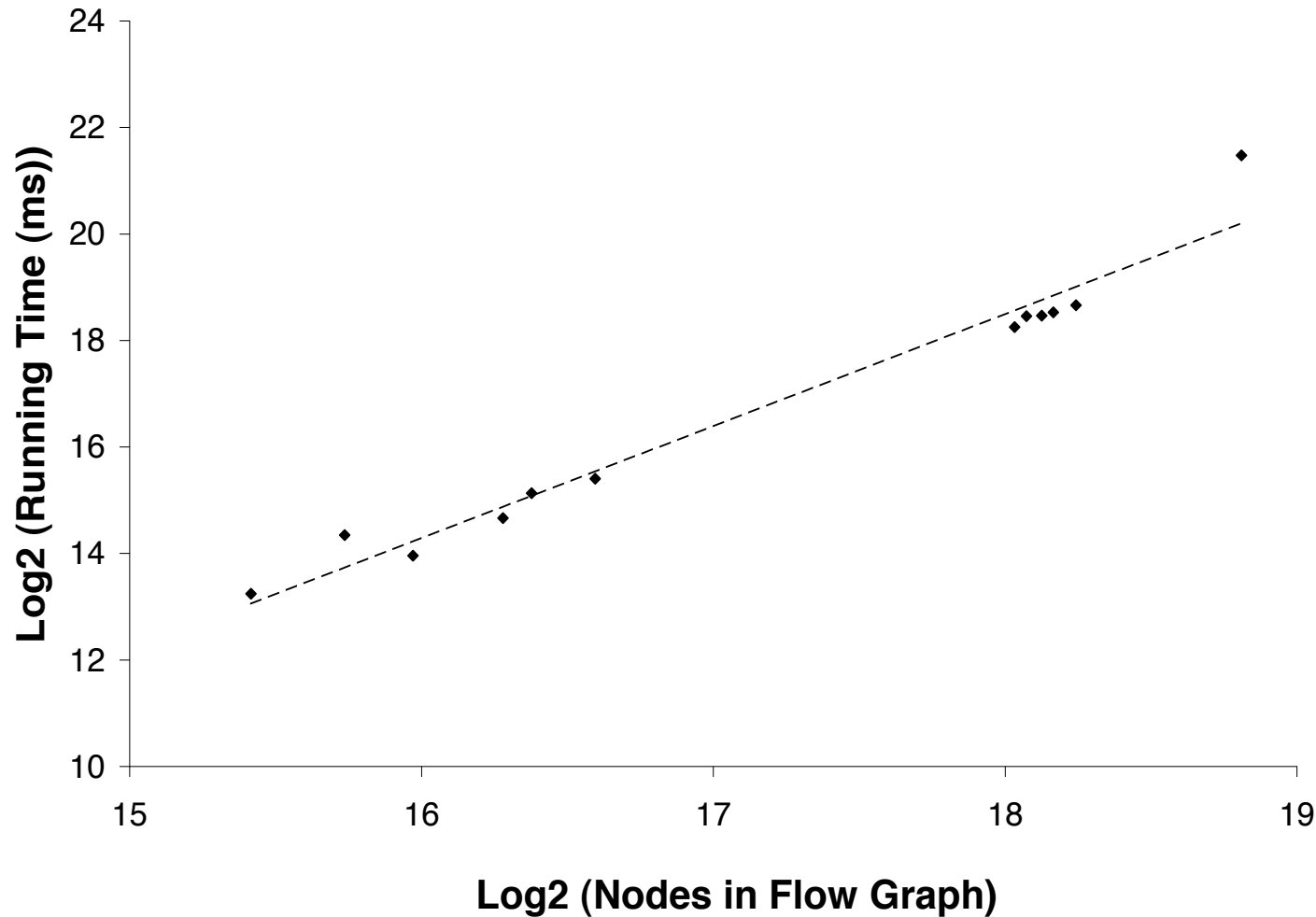
$$E = N^{1.17} \text{ (} N^{1.05} \text{ excluding "outlier");}$$

$$k \leq 4.44$$

## fop “outlier” benchmark

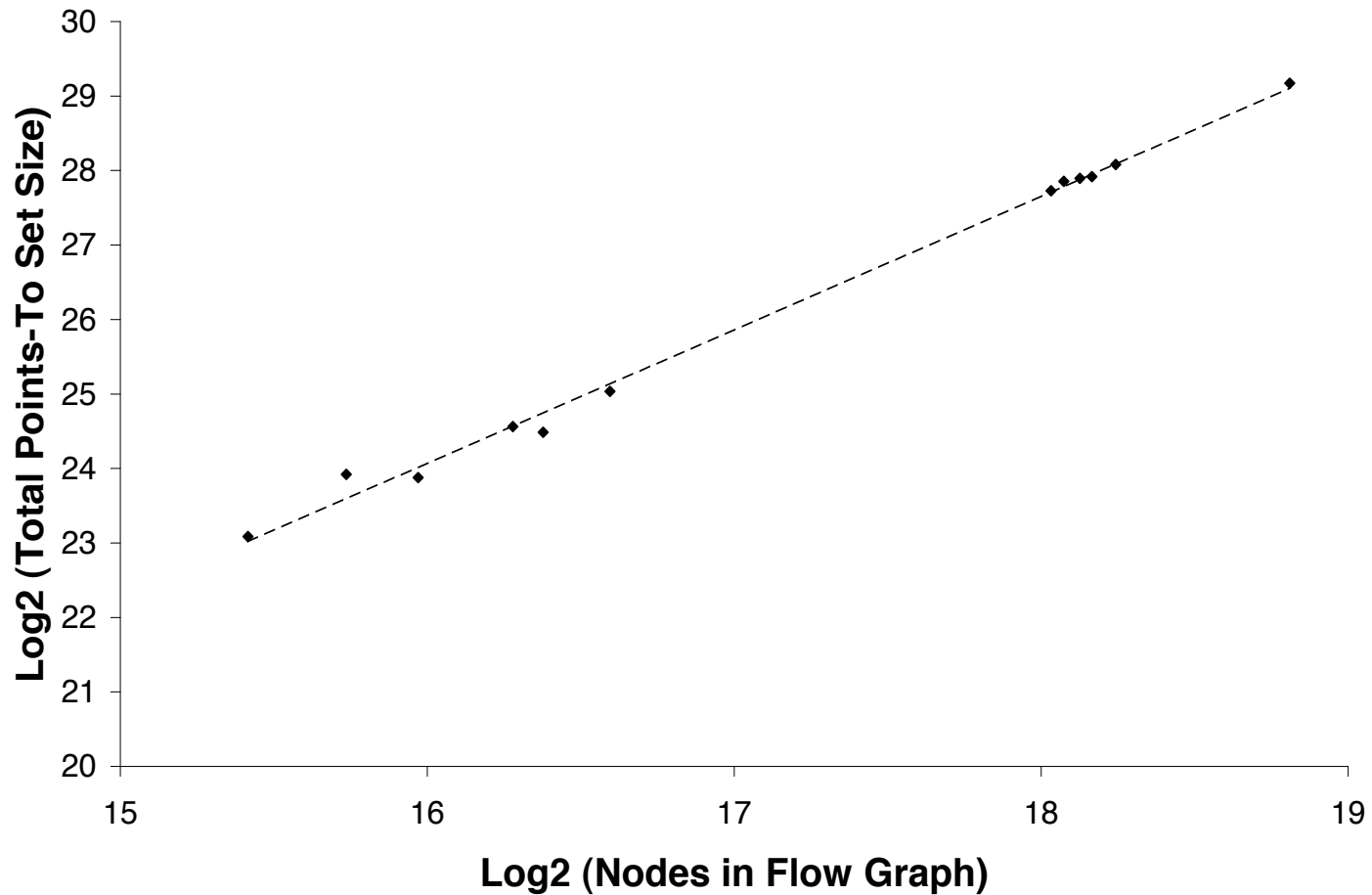
- Extensive use of library array manipulation routines
  - E.g., from `java.util.Arrays`
- Arrays + context-insensitive handling of routines pollutes results
- **Lesson**: targeted context sensitivity could improve both precision and performance
  - Especially for array-handling routines

## Is quadratic scaling observed?



$$\text{Time} = N^{2.10} \text{ (} N^{1.92} \text{ excluding "outlier")}$$

## How tight is quadratic bound?



Total points-to size =  $N^{1.79}$

## Other Factors

- Standard techniques can provide constant-factor speedups
  - Bit vector parallelism, type filters, on-the-fly call graph, preprocessing, cycle elimination
- Space considerations very important in practice
  - May not want exhaustive use of delta sets
  - BDDs / shared bit sets reduce space but complicate running time analysis
- Detailed discussion in paper

## Open questions

- What about other languages?
  - Some evidence that C programs are *not*  $k$ -sparse [PKH03]; may explain greater importance of cycle elimination
  - Result translates to 0-CFA; are functional programs  $k$ -sparse?
- Time complexity for BDDs / shared bit sets?
- Is tighter bound possible?
  - Demand-driven analysis may have less required output
- Does  $k$ -sparsity help other analyses?

## Conclusions

- Andersen's is quadratic for  $k$ -sparse inputs
- Realistic Java programs are  $k$ -sparse
  - **Strong typing**
  - **Encapsulation**
- Explains (partially) the scalability of Andersen's for Java in practice; no cubic bottleneck!

**Thanks!**